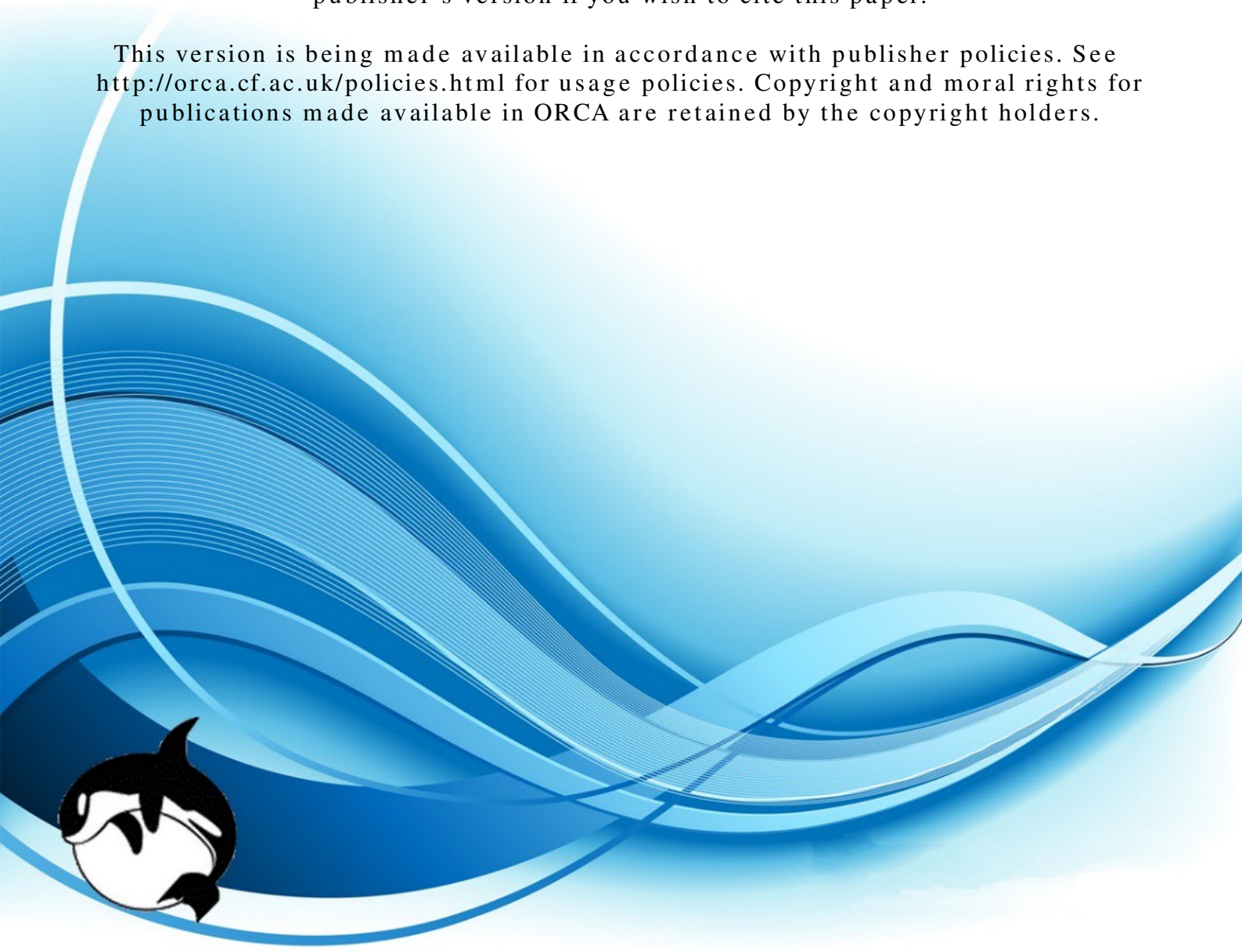


Morton Ordering of 2D Arrays for Efficient Access to Hierarchical Memory

Journal Title
XX(X):1–16

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

David. W. Walker[1]

Abstract

This paper investigates the recursive Morton ordering of two-dimensional arrays as an efficient way to access hierarchical memory across a range of heterogeneous computer platforms, ranging from many-core devices, multi-core processor, clusters, and distributed environments. A brief overview of previous research in this area is given, and algorithms that make use of Morton ordering are described. These are then used to demonstrate the efficiency of the Morton ordering approach by performance experiments on different processors. In particular, timing results are presented for matrix multiplication, Cholesky factorisation, and fast Fourier transform algorithms. The use of the Morton ordering approach leads naturally to algorithms that are recursive, and exposes parallelism at each level of recursion. Thus, the approach advocated in this talk not only provides convenient and efficient access to hierarchical memory, but also provides a basis for exploiting parallelism.

Keywords

Morton ordering; hierarchical memory; parallel algorithms

Introduction

Modern high performance computers are characterised by deeply nested hierarchical memories, and application performance may be significantly degraded if data movement between the different memory layers is not performed efficiently. Maintaining high spatial and/or temporal locality of reference is necessary to reduce data movement overhead and to keep more frequently used data in the higher levels of the memory hierarchy, and may be achieved through compiler transformations, or by the programmer at the source code level. Two and three-dimensional arrays are commonly-used data structures in scientific computing, and are usually stored in memory in one of two canonical layouts: row-major (RM) order or column-major (CM) order. In the case of an $m \times n$ array that is stored contiguously in memory, the address offset in elements from the start of the array of the element at row i and column j is $in + j$ for RM order, and $jm + i$ for CM order. Tiled algorithms are often used to achieve good locality of reference, and hence good performance. A tiled algorithm transforms nested loops by first organising each loop as a loop over blocks of some size, and an inner loop over items in a block. Where algorithmically valid, the resulting loops are then re-ordered so the "block" loops are the outer loops and the loops over items in a block are the inner loops. For a 2D matrix this is equivalent to dividing the matrix into rectangular tiles and acting on each one at a time. Tiled algorithms are expressed in terms of interactions between tiles. For example, in a tiled matrix multiplication algorithm, $C = AB$, each tile of the output matrix can be formed by multiplying a row of tiles of A by a column of tiles of B. Tiling an algorithm changes the execution order of operations and hence the data access pattern. Given a matrix with a canonical layout the aim is to maximise reuse of a tile's data in the higher levels of the memory hierarchy, with the tile size being chosen to match the capacity of some level in the memory hierarchy. Whereas the tiles in a tiled algorithm are all the same size and shape, blocked algorithms are expressed in terms of interactions between blocks in which the blocks do not have to be identical in size. For example, the LAPACK library[1] is largely based on blocked algorithms through the use of BLAS3 operations[5]. A tiled algorithm is a particular

School of Computer Science & Informatics, Cardiff University, Cardiff CF24 3AA, UK

Corresponding author:
David W. Walker, School of Computer Science & Informatics, Cardiff University, 5 The Parade, Cardiff CF24 3AA, UK.
Email: WalkerDW@cardiff.ac.uk

type of blocked algorithm. In a blocked algorithm the same computations are carried as in the unblocked version; however, the order of execution is changed.

For a canonically ordered array, spatial locality can only be exploited with respect to one array dimension. In the absence of an API allowing the programmer to control explicitly the movement of data between the levels of the memory hierarchy, tiled and blocked algorithms for such an array will still not be optimal in managing data movement. The basic idea of the research presented in this paper is that the performance of tiled algorithms can be improved by using non-canonical data orderings, such as space-filling curves and Morton ordering [12]. It is posited that such non-canonical orderings can support efficient access to hierarchical memory across a range of heterogeneous computer platforms, ranging from many-core devices, multi-core processor, clusters, and distributed environments. Morton ordering (see the next section) has been used to optimise database access, in image processing algorithms, and in dense linear algebra computations [15]. The use of Morton ordering in matrix multiplication was also investigated by Valsalam and Skjellum for an earlier generation of processors [16], who considered a number of matrix multiplication algorithms, including Strassen's algorithm. The use of the Hilbert curve and Morton ordering in data layout has been investigated in molecular dynamics applications [11].

This paper makes the following contributions:

1. The use of partial Morton orderings is considered so that the minimum tile size is larger than 2×2 (the case considered by Thiyagalingam et al. [15]).
2. Performance results are presented for a tiled fast Fourier transform (FFT) algorithm, as well as for matrix multiplication and Cholesky factorization, and the optimal tile size for Morton order matrices is investigated.

The use of the Morton ordering approach, and similar approaches based on space-filling curves, leads naturally to algorithms that are recursive, and exposes parallelism at each level of recursion. Thus, the approach advocated in this paper not only provides convenient and efficient access to hierarchical memory, but also provides a basis for exploiting parallelism. Furthermore, good spatial locality is maintained at all levels of the recursion.

Morton Ordering

Morton ordering takes a two-dimensional array stored in row-major order and re-orders it as a 2×2 block array in which the items of each block is stored in row-major order. This process can then be applied recursively to each of the four blocks, and after r levels of recursion the array will be re-ordered as 4^r sub-arrays, each in row-major order. This is illustrated in Fig. 1 for $m = n = 8$, which shows Morton ordering being applied to level $r = 2$, resulting in 16 sub-arrays. A similar approach can be applied to arrays in column-major order, and for arrays of dimension greater than 2.

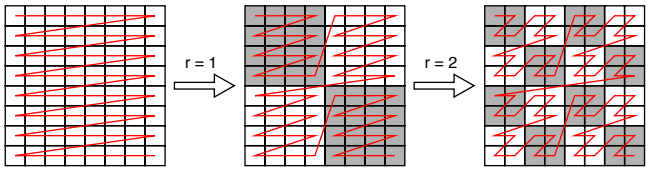


Figure 1. The lefthand part of the figure shows the original array. The middle part of the figure shows the result of Morton ordering to level $r = 1$. The righthand part of the figure shows the result of Morton ordering to level $r = 2$. Each small square represents one array item, and the continuous line between cell centres shows the order in which they are stored, starting in the top left corner. The shading highlights the division into sub-blocks.

Morton ordering can be applied to arbitrary arrays, however, for the rest of this paper attention will focus on Morton ordering of $n \times n$ arrays, where $n = 2^t$. Applying Morton ordering to such an array to level r results in sub-arrays of size $2^{t-r} \times 2^{t-r}$. Level $r = 0$ corresponds to the original array, and so $0 \leq r < t$. If $r = t - 1$ the Morton blocks are of minimum size, namely 2×2.

Applying Morton ordering to a depth r can be expressed as a manipulation of the bitwise representation of the row and column array indices, (i, j), to give the Morton index, k_r. The upper r bits of i are interleaved with the corresponding bits of j to form the upper $2r$ bits of k_r. The lower $t - r$ bits of i form the next least significant bits of k_r, and the lower $t - r$ bits of j form the least significant bits of k_r. This is shown in Fig. 2. The sub-arrays defined by Morton ordering can be numbered consecutively from 0 according to the order in which they are visited. The interleaved upper r bits of i and j give the number of the sub-array containing (i, j), while the lower $t - r$ bits of i and j give the row and column index within the sub-array. For example, consider position $(2, 3)$ in an 8×8 matrix, which corresponds to location $k = 2 * 8 + 3 = 19$ in a row-major ordering. Applying one level of Morton ordering ($r = 1$) this item would be at index $(001011)_2 = 11$. The upper 2 bits indicate that the item in in sub-array 0. Applying a second level ($r = 2$) gives an index of $(001101)_2 = 13$, where the upper 4 bits indicate that the item is in sub-array 3. This example can be verified using Fig. 1.

Methods for converting between canonical and Morton ordering based on dilated integers have been investigated by Raman and Wise[17]. The dilated form of an integer is obtained by interposing a 0 between each of its bits. For example, consider $i = 13$ so that $i = (1101)_2$: then the dilated form of i is $d(i) = (01010001)_2 = 81$. The bits of two integers, i and j, can be interleaved by forming $2d(i) + d(j)$.

Given a level $r-1$ Morton ordering, a level r Morton ordering can be achieved by cyclically rotating bits $t-r$ to $2(t-r)$ of the index one position to the right. To go from a level r to a level $r-1$ Morton ordering it is simply necessary to cyclically rotate the same set of bits one position to the left. Returning to the example above, for which item $(2,3)$ of an 8×8 array has index 19 in a row-major ordering, then to go from level 0 to level 1 of the Morton ordering requires bits $t-r=2$ to $2(t-r)=4$ to be cyclically rotated to the right. Since $19 = (010011)_2$, cyclically rotating bits 2 to 4 one step to the right gives $(001011)_2 = 11$. To go from level 1 to level 2 requires bits 1 to 2 to be cyclically shifted to the right (i.e., exchanged), giving $(001101)_2 = 13$.

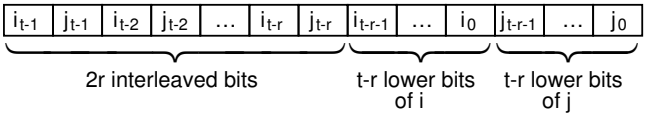


Figure 2. The bits of the Morton index k_r at level $r \geq 1$.

Morton ordering can also be represented in terms of linear algebra operations. Suppose X_0 is an $n \times n$ matrix, where $n = 2^t$. Let x_0 be the row vector created by concatenating the rows of X_0, i.e., x_0 represents X_0 stored in row-major order. Let X_r denote the matrix obtained by applying Morton ordering to level r to X_0, and let x_r be the corresponding vector of concatenated rows. Then,

$$x_r = x_{r-1}(I_p \otimes (\Pi_{2b} \otimes I_b)) \tag{1}$$

where $b = 2^{t-r}$, $p = 2^{2r-1}$, and $1 \leq r < t$. Π_m is an $m \times m$ permutation matrix such that $v\Pi_m^T$ performs a perfect shuffle operation on the elements of the row vector, v, and $A \otimes B$ denotes the Kronecker product of the matrices A and B.

An $n \times n$ array, A, in row-major order with $n = 2^t$, can be re-ordered as a level 1 Morton order array, consisting of four $n/2 \times n/2$ sub-arrays A_{00}, A_{01}, A_{10} and A_{11}, by a simple in-place algorithm. Consider row k of A, where $0 \leq k < n/2$, which consists of the $n/2$ elements that form row k of A_{00} followed by the $n/2$ elements that form row k of A_{01}. Thus, if we denote row k of A_{00} and A_{01} by a_k and b_k, respectively, then the first k rows of A are laid out as follows in linearised index space:

$$a_0 b_0 a_1 b_1 \ldots a_{n/2-1} b_{n/2-1}. \tag{2}$$

Level 1 Morton ordering transforms this layout by means of an unshuffle operation (see next subsection) to give the ordering:

$$a_1 a_2 \ldots a_{n/2-1} b_0 b_1 \ldots b_{n/2-1} \tag{3}$$

A recursive algorithm for converting a power-of-two array from row-ordered to a level r Morton order using unshuffle operations is shown in Alg. 1. It is a simple matter to extend this algorithm to more general arrays.

ALGORITHM 1: morton: recursive routine for transforming row-ordered power-of-two array to level r Morton order.

Function `morton` (*A,n,r*)
  Input: Array A of size $n \times n$, integer $n = 2^t$, and integer r $(0 \leq r < t)$ for terminating recursion.
  Output: The array A in level r Morton order.
  if $(r \leq 0)$ **then**
    return
  end
  $n1 = n/2$
  `unshuffle` $(A00, n1, n1, n1)$
  `unshuffle` $(A10, n1, n1, n1)$
  `morton` $(A00, n1, r-1)$
  `morton` $(A01, n1, r-1)$
  `morton` $(A10, n1, r-1)$
  `morton` $(A11, n1, r-1)$
  return
end

Unshuffle and Shuffle Operations

Consider the following contiguous sequence of $2m$ row vectors: $a_1 b_1 a_2 b_2 \ldots a_m b_m$, where each a_i is itself a contiguous vector of ℓ_a elements, and each b_i is a contiguous vector of ℓ_b elements. Then the unshuffle operation corresponds to the following reordering of the vectors:

$$a_1 b_1 a_2 b_2 \ldots a_m b_m \rightarrow a_1 a_2 \ldots a_m b_1 b_2 \ldots b_m \tag{4}$$

The shuffle operation performs the inverse of this reordering:

$$a_1 a_2 \ldots a_m b_1 b_2 \ldots b_m \rightarrow a_1 b_1 a_2 b_2 \ldots a_m b_m \tag{5}$$

In the algorithms presented in this paper, use is made of the `shuffle` and `unshuffle` functions, in which the first argument is (a pointer to) the data to be reordered, and the subsequent arguments are ℓ_a, ℓ_b, and m. The shuffle and unshuffle operations can themselves be expressed recursively[9].

Matrix Multiplication

Suppose the matrices A and B are multiplied to give the matrix C, where all matrices are of size $n \times n$ where $n = 2^t$. This matrix multiplication can be expressed in block form as:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$
$$= \left[\begin{array}{c|c} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ \hline A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{array}\right]$$

where each of the blocks is of size $n/2 \times n/2$. This shows how the product of two matrices can be expressed in terms of the products of smaller matrices, and can be used as the basis of a recursive algorithm for matrix multiplication. At each level of the recursion the matrices to be multiplied are partitioned into quadrants. The recursion terminates at some specified depth, $r < t$. If the matrices A and B are Morton ordered then only contiguously stored matrix blocks are multiplied and the recursive matrix multiplication algorithms can be expressed as in Alg. 2, where the routine `matmul` can be any general-purpose matrix multiplication routine, such as `DGEMM` from the LAPACK library [1].

ALGORITHM 2: mmRecursive: Recursive matrix multiplication of power-of-two matrices. Routine `matmul` performs $C \leftarrow C + AB$.

Function `mmRecursive` (*A,B,C,n,r*)
  Input: Matrices A, B, and C of size $n \times n$, integer $n = 2^t$, integer r $(0 \leq r < t)$ for terminating recursion. All elements of matrix C must be zero initially,
  Output: On exit, the matrix C contains AB.
  if $(r \leq 0)$ **then**
    `matmul` (A, B, C, n)
  else
    $n1$=$n/2$
    `mmRecursive` $(A00, B00, C00, n1, r-1)$
    `mmRecursive` $(A01, B10, C00, n1, r-1)$
    `mmRecursive` $(A00, B01, C01, n1, r-1)$
    `mmRecursive` $(A01, B11, C01, n1, r-1)$
    `mmRecursive` $(A10, B00, C10, n1, r-1)$
    `mmRecursive` $(A11, B10, C10, n1, r-1)$
    `mmRecursive` $(A10, B01, C11, n1, r-1)$
    `mmRecursive` $(A11, B11, C11, n1, r-1)$
  end
end

Algorithm 2 is a tiled algorithm as all the computation involves the multiplication of tiles of size $2^{t-r} \times 2^{t-r}$ in the leaves of the recursion tree. It should also be noted that, if A and B are Morton order matrices, then Alg. 2 will leave the output matrix C in Morton order.

Cholesky Factorisation

Cholesky factorisation decomposes a real, symmetric, diagonally-dominant matrix A as LL^T, where L is a lower triangular matrix. The blocked Cholesky factorisation algorithm is based on the following matrix partitioning in which A_{00} is $b \times b$, $A_{10} = A_{01}^T$ is $(n-b) \times b$, and A_{11} is $(n-b) \times (n-b)$:

$$\left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] = \left[\begin{array}{c|c} L_{00} & 0 \\ \hline L_{10} & L_{11} \end{array}\right] \left[\begin{array}{c|c} L_{00}^T & L_{10}^T \\ \hline 0 & L_{11}^T \end{array}\right]$$
$$= \left[\begin{array}{c|c} L_{00}L_{00}^T & L_{00}L_{10}^T \\ \hline L_{10}L_{00}^T & L_{10}L_{10}^T + L_{11}L_{11}^T \end{array}\right]$$

The recursive right-looking Cholesky factorisation algorithm is shown in Alg. 3, in which routine `cholesky` performs a Cholesky factorisation on a $b \times b$ block; `triangularSolve` solves $L_{10}L_{00}^T = A_{10}$; `symmetricRankUpdate` performs a symmetric rank-b update on A_{11}, replacing it with $A_{11} - L_{10}L_{10}^T$. Algorithm 3 exhibits tail recursion, but as noted by Gustavson [8], the algorithm can also be cast in binary recursive form, as shown in Alg. 4

ALGORITHM 3: choleskyTailRecursive: Tail recursive Cholesky factorisation of real symmetric matrix.

Function `choleskyTailRecursive` (*A,n,b*)
  Input: Real symmetric matrix A of size $n \times n$, integer n, integer b is the block size.
  Output: On exit, the lower-triangular part of matrix A contains the Cholesky factor, L.
  if $(n = b)$ **then**
    `cholesky` (A, b)
  else
    `cholesky` $(A00, b)$
    `triangularSolve` $(A10, A00, n-b, b)$
    `symmetricRankUpdate` $(A11, A10, n-b, b)$
    `choleskyTailRecursive` $(A11, n-b, b)$
  end
end

Fast Fourier Transform

The discrete Fourier transform (DFT) of a two-dimensional array, X, of size $n \times n$ is given by:

$$y_{jk} = \sum_{p=0}^{n-1} \sum_{q=0}^{n-1} x_{pq} \exp\left(-2\pi i(jp + kq)/n\right) \qquad (6)$$

where $i = \sqrt{-1}$ and $0 \leq j, k < n$. x_{pq} is the element in row p and column q of X, and this is also indicated by writing $X = (x_{pq})$. The 2D DFT can be expressed in terms of matrices as:

$$Y = F_n X F_n \qquad (7)$$

ALGORITHM 4: choleskyBinaryRecursive: Binary recursive Cholesky factorisation of real symmetric matrix. $A00$, $A01$, $A10$ and $A11$ are the four quadrants of the input matrix A, and are all $n/2 \times n/2$ matrices.

Function `choleskyBinaryRecursive` (*A,n,b*)
  Input: Real symmetric matrix A of size $n \times n$, integer n, integer b is the block size at which the recursion terminates.
  Output: On exit, the lower-triangular part of matrix A contains the Cholesky factor, L.
  if $(n = b)$ **then**
    `cholesky` (A, n, b)
  else
    `choleskyBinaryRecursive` $(A00, n/2, b)$
    `triangularSolve` $(A10, A00, n/2, n/2)$
    `symmetricRankUpdate` $(A11, A10, n/2, n/2)$
    `choleskyBinaryRecursive` $(A11, n/2, b)$
  end
end

where $F_n = (\omega_n^{pq})$ and $w_n = \exp(-2\pi i/n)$. It should be noted that the complex matrix F_n is symmetric: $F_n = F_n^T$.

It is well known that the fast Fourier transform (FFT) replaces the dense matrix multiplications in Eq. 7 by a series of sparse matrix multiplications (for example, see Van Loan[10]). Thus, when $n = 2^t$, Eq. 7 may be written as,

$$\begin{aligned} Y = F_n X F_n &= F_n X F_n^T \\ &= A_t \ldots A_1 P_n^T X P_n A_1^T \ldots A_t^T \end{aligned} \quad (8)$$

where P_n^T is an $n \times n$ permutation matrix that, when applied to a column vector v, stores v_j at the index obtained by reversing the t bits of j. Thus, $P_n^T X P_n$ re-orders the rows and columns of X by bit-reversing the row and column indices. In addition,

$$A_q = I_r \otimes B_L \quad (9)$$

$$B_L = \begin{bmatrix} I_{L_*} & \Omega_{L_*} \\ I_{L_*} & -\Omega_{L_*} \end{bmatrix} \quad (10)$$

$$\Omega_{L_*} = \mathrm{diag}(1, \omega_L, \ldots, \omega_L^{L_*-1}) \quad (11)$$

where I_m is the $m \times m$ identity matrix, $L = 2^q$, $r = n/L$, $L_* = L/2$. Equations 8-11 express the Cooley-Tukey radix-2 formulation of the FFT algorithm.

Common 2D FFT Algorithms

There are two common approaches to evaluating a 2D FFT based on Eq. 8, which are presented here for clarity of exposition.

1. The first algorithm is shown in Alg. 5, and will be referred to as the *transpose FFT algorithm*. This algorithm performs all the pre-multiplications of X in Eq. 8 to give $\tilde{X} = A_t \ldots A_1 P_n^T X$ and then transposes Eq. 8 to give $Y^T = A_t \ldots A_1 P_n^T \tilde{X}^T$. The pre-multiplications done here on $\tilde{X}^T$ are identical to those performed on X in the first stage of the algorithm, and correspond to performing 1D FFTs along the n rows of the matrix. Having performed the second set of multiplications the result is transposed to give Y. This approach separates out the operations on the rows and columns of X, and maintains unit stride when X is stored in row-major order. For column-major matrices unit stride access is achieved by doing all the post-multiplications first. Step q of each pre-multiplication stage can be expressed as:

$$X \leftarrow A_q X \quad (12)$$

2. An alternative approach is to operate on both rows and columns in each stage of the algorithm, as shown in Alg. 6, which is referred to as the *vector-radix FFT algorithm* in Van Loan[10]. Step q of Alg. 6 can be expressed as:

$$X \leftarrow A_q X A_q^T \quad (13)$$

ALGORITHM 5: transposeFFT: 2D FFT with transpose.

Function `transposeFFT` (*X,n*)
  Input: Matrix X of size $n \times n$.
  Output: The matrix X is overwritten by its Fourier transform.
  $t = \log_2 n$
  $X = P_n^T X P_n$
  for $q = 1, 2, \ldots, t$ **do**
    $X = A_q X$
  end
  `transpose` (X, n)
  for $q = 1, 2, \ldots, t$ **do**
    $X = A_q X$
  end
  `transpose` (X, n)
end

ALGORITHM 6: vectorradixFFT: Vector-radix 2D FFT.

Function `vectorradixFFT` (*X,n*)
  Input: Matrix X of size $n \times n$.
  Output: The matrix X is overwritten by its Fourier transform.
  $t = \log_2 n$
  $X = P_n^T X P_n$
  for $q = 1, 2, \ldots, t$ **do**
    $X = A_q X A_q^T$
  end
end

From Eq. 9 it may be seen that A_q is a block-diagonal matrix in which each of the r diagonal blocks is B_L. Partitioning X in the same way as B_L, i.e., as a 2×2 block matrix with blocks of size $L_* \times L_*$, then Eq. 12 of Alg. 5 can be written as

$$X_{ij} \leftarrow B_L X_{ij} \quad (14)$$

which gives,

$$X_{ij} \equiv \left[\begin{array}{c|c} X_{ij}^{00} & X_{ij}^{01} \\ \hline X_{ij}^{10} & X_{ij}^{11} \end{array}\right] \leftarrow \left[\begin{array}{c|c} X_{ij}^{00} + \Omega_{L_*} X_{ij}^{10} & X_{ij}^{01} + \Omega_{L_*} X_{ij}^{11} \\ \hline X_{ij}^{00} - \Omega_{L_*} X_{ij}^{10} & X_{ij}^{01} - \Omega_{L_*} X_{ij}^{11} \end{array}\right] \quad (15)$$

where the superscripts 00, 01, 10, and 11 refer to the upper-left, upper-right, lower-left and lower-right quadrants of X_{ij}.

Applying a similar approach to Eq. 13 of Alg. 6 shows that the blocks of X are updated at each stage of the algorithm as follows:

$$X_{ij} \leftarrow B_L X_{ij} B_L^T \quad (16)$$

The product $B_L X_{ij}$ may be computed as in Eq. 15, and the result is then post-multiplied by B_L^T, which further updates the blocks of X:

$$\left[\begin{array}{c|c} X_{ij}^{00} & X_{ij}^{01} \\ \hline X_{ij}^{10} & X_{ij}^{11} \end{array}\right] \leftarrow \left[\begin{array}{c|c} X_{ij}^{00} + X_{ij}^{01}\Omega_{L_*} & X_{ij}^{01} - X_{ij}^{01}\Omega_{L_*} \\ \hline X_{ij}^{10} + X_{ij}^{11}\Omega_{L_*} & X_{ij}^{10} - X_{ij}^{11}\Omega_{L_*} \end{array}\right] \quad (17)$$

For row-ordered matrices, the updates in Eq. 15 can be done with unit stride accesses to X. Unit stride access can also be maintained in the updates in Eq. 17 by transposing X before and after the second set of updates. However, this requires two transpositions to be performed in each of the t stages of Alg. 6, which may result in excessive data movement.

Recursive 2D FFTs

Both of the algorithms presented above involve loops over $q = 1, \ldots, t$. For each value of q, blocks of X of size $2^q \times 2^q$ are updated using the four constituent sub-blocks, according to Eqs. 15 and 17. The number of rows and columns in each block doubles for successive values of q. This means that the 2D FFT can readily be performed by a recursive algorithm when the matrix X is stored in Morton order.

To formulate a recursive version of the 2D FFT algorithms presented in Algs. 5 and 6 for Morton order arrays the algorithm should terminate the recursion at some level in the recursion tree when the block size is $b = 2^s$ for $0 < s < t$. A 2D FFT is performed on each of the $b \times b$ blocks of X, using any algorithm, and the algorithm then moves back up the recursion tree, first assembling $2b \times 2b$ blocks, and then $4b \times 4b$ blocks, and so on. To develop the recursive algorithm the following three lemmas are required:

- *Lemma 1*: The radix-2 splitting equation (Theorem 1.2.1 from Van Loan[10])

$$F_n \Pi_n = B_n (I_2 \otimes F_{n/2}) \quad (18)$$

  where Π_n is an $n \times n$ matrix such that $\Pi_n v$ performs a perfect shuffle operation on the elements of the column vector, v.
- *Lemma 2*: $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$, if the matrix multiplications AC and BD are defined.
- *Lemma 3*: $I_p \otimes (I_q \otimes A) = I_{pq} \otimes A$.

It should be noted that Lemma 2 and Lemma 3 correspond to the properties of the Kronecker product referred to as Kron1 and Kron7, respectively, in Van Loan[10]. The following theorem provides the basis for a recursive 2D FFT algorithm.

Theorem 1. *If* $1 \le b \le n$ *then*

$$F_n \Pi_{b,n} = B_{b,n} (I_{n/b} \otimes F_b) \quad (19)$$

where

$$\begin{aligned} \Pi_{b,n} &= \Pi_n (I_2 \otimes \Pi_{n/2})(I_4 \otimes \Pi_{n/4}) \ldots (I_{n/(2b)} \otimes \Pi_{2b}) \\ B_{b,n} &= B_n (I_2 \otimes B_{n/2})(I_4 \otimes B_{n/4}) \ldots (I_{n/(2b)} \otimes B_{2b}) \end{aligned} \quad (20)$$

Proof. Proof is by induction on b. Equation 19 holds when $b = n$ since in this case $\Pi_{b,n} = \Pi_n$ and $B_{b,n} = B_n$ and Eq. 18 is recovered. Now suppose Eq. 19 is true for $b = 2\beta$ for $1 \le \beta \le n/2$. Then,

$$\begin{aligned} F_n \Pi_{\beta,n} &= F_n \Pi_{2\beta,b} (I_{n/(2\beta)} \otimes \Pi_{2\beta}) \\ &= B_{2\beta,n} (I_{n/(2\beta)} \otimes F_{2\beta})(I_{n/(2\beta)} \otimes \Pi_{2\beta}) \\ &= B_{2\beta,n} (I_{n/(2\beta)} \otimes (F_{2\beta} \Pi_{2\beta})) \text{ by Lemma 2} \\ &= B_{2\beta,n} (I_{n/(2\beta)} \otimes (B_{2\beta} (I_2 \otimes F_\beta)) \text{ by Lemma 1} \\ &= B_{2\beta,n} (I_{n/(2\beta)} \otimes B_{2\beta})(I_{n/(2\beta)} \otimes (I_2 \otimes F_\beta)) \\ &\qquad \text{by Lemma 2} \\ &= B_{2\beta,n} (I_{n/(2\beta)} \otimes B_{2\beta})(I_{n/\beta} \otimes F_\beta) \\ &\qquad \text{by Lemma 3} \\ &= B_{\beta,n} (I_{n/\beta} \otimes F_\beta) \end{aligned}$$

Thus, Eq. 19 is also true for $b = \beta$, which completes the inductive proof.

Using Eq. 19 the 2D FFT of X may be written,

$$\begin{aligned} F_n X F_n &= F_n X F_n^T \\ &= B_{b,n} (I_{n/b} \otimes F_b) H_{b,n} (I_{n/b} \otimes F_b) B_{b,n}^T \end{aligned} \quad (21)$$

where $H_{b,n} = \Pi_{b,n}^T X \Pi_{b,n}$. Now, $(I_{n/b} \otimes F_b) H_{b,n} (I_{n/b} \otimes F_b)$ is the result of partitioning the matrix $H_{b,n}$ into $b \times b$ blocks and independently performing a 2D FFT on each block. The matrix $I_{n/k} \otimes \Pi_k$ is a permutation matrix such that $(I_{n/k} \otimes \Pi_k)v$ performs a perfect shuffle on blocks of the column vector v of size n/k. This is equivalent to cyclically rotating the lower p bits of the vector index one step to the left, where $k = 2^p$. Repeating such operations, as in Eq. 20, it may be seen that $\Pi_{b,n}$ is a permutation matrix that performs a partial bit reversal, i.e., if $w = \Pi_{b,n} v$ then $w_{j'} = v_j$ where j' is the partial bit reversal of j. If the bits of j are,

$$(j)_2 = j_{t-1} j_{t-2} \cdots j_1 j_0 \tag{22}$$

where j_0 is the least significant bit, then,

$$(j')_2 = j_{s-1} \cdots j_1 j_0 j_s j_{s+1} \cdots j_{t-2} j_{t-1} \tag{23}$$

A recursive algorithm for performing partial bit-reversal on a vector is shown in Alg. 7. This makes use of a shuffle operation, introduced above, in which the vectors being shuffled are all of length 1.

ALGORITHM 7: PBR: Recursive routine for performing partial bit-reversal on a vector. $x \leftarrow \Pi_{b,n} x$.

Function `PBR` (*x,n,b*)
  Input: Vector x of length n, integer $n = 2^t$, and integer $b = 2^s$.
  Output: The vector x in partially bit-reversed order.
  if *(*$n <= b$*)* **then**
    return
  end
  `PBR` $(x, n/2, b)$
  `PBR` $(x + n/2, n/2, b)$
  `shuffle` $(x, 1, 1, n/2)$
end

Note that when $s = 0$ then $b = 1$, and $\Pi_{1,n} = P_n$, corresponding to a complete bit reversal. Equation 21 may be written,

$$F_n X F_n = F_n X F_n^T = A_t \ldots A_{s+1} K_{b,n} A_{s+1}^T \ldots A_t^T \tag{24}$$

where $K_{b,n} = (I_{n/b} \otimes F_b) H_{b,n} (I_{n/b} \otimes F_b)$.

Equation 24 shows how to modify Algs. 5 and 6 to give corresponding recursive algorithms in which the recursion terminates at block size $b \times b$. The algorithm shown in Algs. 8 and 9 is the recursive version of Alg. 5, and that shown in Algs. 10 and 11 is the recursive version of Alg. 6. In Algs. 9 and 11 the routine `FFT2D` performs a 2D FFT on the matrix X of size $b \times b$, overwriting the input with the result. Algorithm 8 calls routine `recursiveFFT` two times, so the recursion tree is traversed twice. The first call evaluates $K_{b,n}$ (see Eq. 24) in each of the leaf nodes of the recursion tree, and then calls routine `butterflyPre` to apply the butterfly operations in Eq. 15 at each non-leaf node to give:

$$Y_{b,n} = A_t \ldots A_{s+1} K_{b,n} \tag{25}$$

This is then transposed and used to evaluate $A_t \ldots A_{s+1} Y_{b,n}^T$ in the second call to `recursiveFFT`. Transposing the result of this gives the required result. As in Alg. 5, this approach ensures unit stride access when performing the butterfly operations. However, in Alg. 11 the butterfly operations in Eqs. 15 and 17 are both applied in the same node of the recursion tree, which is traversed only once. Unit stride access is not maintained in the post-multiplicative butterfly operations in Eq. 17. This will result in more data movement, particularly in higher levels of the recursion tree, but this is offset by avoiding the data movement in the transposition operations.

ALGORITHM 8: transposeFFT2: FFT using recursion and block size b.

Function `transposeFFT2` (*X,n,b*)
  Input: Matrix X of size $n \times n$, integer n, integer block size b.
  Output: The matrix X is overwritten by its Fourier transform.
  $X = \Pi_{b,n}^T X \Pi_{b,n}$
  `recursiveFFT` $(X, n, b, 1)$
  `transpose` (X, n)
  `recursiveFFT` $(X, n, b, 0)$
  `transpose` (X, n)
end

ALGORITHM 9: recursiveFFT: recursive FFT with block size b.

Function `recursiveFFT` (*X,n,b,dofft*)
  Input: Matrix X of size $n \times n$, integer n, integer termination size b, boolean *dofft*.
  Output: The matrix X is overwritten by its partial Fourier transform, or the full transform if $L = b$.
  if *(*n==b*)* **then**
    if (*dofft*) `fft2D` (X, b)
  else
    $n2 = n/2$
    `recursiveFFT` $(X00, n2, b,$ *dofft*$)$
    `recursiveFFT` $(X01, n2, b,$ *dofft*$)$
    `recursiveFFT` $(X10, n2, b,$ *dofft*$)$
    `recursiveFFT` $(X11, n2, b,$ *dofft*$)$
    `butterflyPre` (X, n)
  end
end

ALGORITHM 10: vectorradixFFT2: Vector-radix FFT using recursion and block size b.

Function `vectorradixFFT2` (*X,n,b*)
  Input: Matrix X of size $n \times n$, integer n, integer block size b.
  Output: The matrix X is overwritten by its Fourier transform.
  $X = \Pi_{b,n}^T X \Pi_{b,n}$
  `recursiveVRFFT` (X, n, b)
end

ALGORITHM 11: recursiveVRFFT: recursive vector-radix FFT.

Function `recursiveVRFFT` (X, n, b)
  Input: Matrix X of size $n \times n$, integer n, integer termination size b.
  Output: The matrix X is overwritten by its partial Fourier transform, or the full transform if $L = b$.
  if $(n==b)$ **then**
    `fft2D` (X, b)
  else
    $n2 = n/2$
    `recursiveVRFFT` $(X00, n2, b)$
    `recursiveVRFFT` $(X01, n2, b)$
    `recursiveVRFFT` $(X10, n2, b)$
    `recursiveVRFFT` $(X11, n2, b)$
    `butterflyPre` (X, n)
    `butterflyPost` (X, n)
  end
end

Recursive FFTs of Morton Order Matrices

In the matrix multiply algorithm shown in Alg. 2 all the work is performed in the leaf nodes of the recursion tree. If the input matrices are stored in Morton order this has no effect on the computation, and Alg. 2 will work correctly provided the matrices multiplied by the routine `matmul` in the leaf nodes are row-major order blocks (or column-major order blocks if that is what `matmul` expects). However, for the FFT algorithms in Algs. 8-11 there are three types of operation associated with non-leaf nodes: (1) partial bit reversal, $X \leftarrow \Pi_{b,n}^T X \Pi_{b,n}$; (2) butterfly computations; and (3) matrix transposition. The algorithms for these operations have to be modified if the matrices are stored in Morton order.

Partial Bit Reversal. A partial bit reversal can be performed on a matrix either by bitwise manipulations or by matrix operations. The bitwise approach for performing partial bit reversal on a Morton ordered matrix is as follows:

1. Find the index k' such that after converting the matrix to Morton order $x_{k'}$ is now stored at index k. The bits of k' are:

$$k_{2t-1}, k_{2t-3}, \ldots, k_{2t-2r+1} | k_{2t-2r-1}, \ldots, k_{t-r} | k_{2t-2}, k_{2t-4}, \ldots, k_{2t-2r} | k_{t-r-1}, \ldots, k_0$$

2. Let k'' be the index at which the element at index k' of the row-major order matrix is stored after the Morton order and partial bit reversal operations. From the bitwise transformations that occur in these operations the bits of k'' are given by:

$$k'_t, k'_0, k'_{t+1}, k'_1 \ldots, k'_{t+r-1}, k'_{r-1} | k'_{2t-1} \ldots, k'_{t+r} | k'_{t-1}, \ldots, k'_r$$

3. Store the element at index k in the Morton order matrix at index k''.

The matrix approach is based on the observation that $\Pi_{b,n}^T X \Pi_{b,n} = (\Pi_{b,n}^T (\Pi_{b,n}^T X)^T)^T$, and that $\Pi_{b,n}^T X$ can be evaluated as shown in Alg. 12, in which the `unshuffle` routine performs an unshuffle permutation on the rows of X. Thus, to evaluate $\Pi_{b,n}^T X$ when X is stored in Morton order requires a Morton order version of the unshuffle operation.

ALGORITHM 12: pbrMorton: recursive routine for evaluating $X \leftarrow \Pi_{b,n}^T X$ for a Morton ordered matrix.

Function `pbrMorton` (X, n, b)
  Input: Matrix X of size $n \times n$, integer $n = 2^t$, and integer $b = 2^s$.
  Output: The matrix $\Pi_{b,n}^T X$.
  if $(n > b)$ **then**
    `unshuffle` $(X, n, n, n/2)$
    `pbrMorton` $(X00, n/2, b)$
    `pbrMorton` $(X01, n/2, b)$
    `pbrMorton` $(X10, n/2, b)$
    `pbrMorton` $(X11, n/2, b)$
  end
end

If the column vector x is formed of the concatenated rows of the $n \times n$ matrix, X, then the partial bit reverse over rows of X can be represented in terms of x as:

$$\mathcal{L}_{n,n} \ldots \mathcal{L}_{2b,n} \mathcal{L}_{b,n} x$$

where $\mathcal{L}_{b,n} = I_{n/b} \otimes (\Pi_b \otimes I_n)$.

Butterfly Operations. When pre- or post-multiplying a Morton ordered matrix by a butterfly matrix it is necessary to process each element in the upper-left quadrant by row (if pre-multiplying) or by column (if post-multiplying). Each such element is updated, together with the corresponding elements in the other three quadrants. The Morton butterfly algorithm follows the row-major version, but for each row and column index, (j, i), it is necessary to find the index k of the corresponding element in the Morton ordered matrix, as follows:

$$(k)_2 = j_{n-1}, i_{n-1}, j_{n-2}, i_{n-2} \ldots, j_{n-r}, i_{n-r} | j_{n-r-1} \ldots, j_0 | i_{n-r-1}, \ldots, i_0$$

Transposition. A matrix stored in Morton form can be transposed recursively as shown in Alg. 13. In this algorithm `transpose` performs a standard matrix transpose, and `exchange` swaps the upper-right and lower-left quadrants. The algorithm is based on the observation that:

$$X = \left[\begin{array}{c|c} X_{00} & X_{01} \\ \hline X_{10} & X_{11} \end{array}\right] \Rightarrow X^T = \left[\begin{array}{c|c} X_{00}^T & X_{10}^T \\ \hline X_{01}^T & X_{11}^T \end{array}\right]$$

ALGORITHM 13: transposeMorton: matrix transpose of a Morton ordered matrix.

Function `transposeMorton` (X, n, b)
  Input: Matrix X of size $n \times n$, integer n, integer block size b.
  Output: The matrix X is overwritten by its transpose.
  if *(n==b)* **then**
    `transpose` (X, b)
  else
    `exchange` $(X01, X10, n/2)$
    `transposeMorton` $(X00, n/2, b)$
    `transposeMorton` $(X01, n/2, b)$
    `transposeMorton` $(X10, n/2, b)$
    `transposeMorton` $(X11, n/2, b)$
  end
end

A Variant of the Algorithm

If v is a column vector, the product $\Pi_{b,n}^T v$ permutes v through a partial bit reversal into blocks of length b, with the elements in each block being of the form $v_{i+k(n/b)}$ for $k = 0, 1, \ldots, b-1$. The n/b blocks are then permuted in bit-reverse order. This can be verified from Eq. 23. The block-based bit reversal can be removed by multiplying $\Pi_{b,n}$ by $(P_m \otimes I_b)$, where $m = n/b$ and P_m is the $m \times m$ bit-reversal matrix. This results in the same blocks of v, but now the first block is $v(0 : b : m-1)$, the second block is $v(1 : b : m)$, and so on. The permutation that gives this ordering is:

$$\mathcal{P}_{m,n} = \Pi_{b,n}(P_m \otimes I_b). \tag{26}$$

If, in addition:

$$\mathcal{B}_{m,n} = B_{b,n}(P_m \otimes I_b), \tag{27}$$

then the properties of the Kronecker product allow Eq. 19 to be written as:

$$F_n \mathcal{P}_{m,n} = \mathcal{B}_{m,n}(I_m \otimes F_b) \tag{28}$$

which is the standard radix-m splitting equation (see section 2.1.3 of Van Loan[10]). This allows the 2D Fourier transform to be expressed as:

$$\begin{aligned} F_n X F_n &= F_n X F_n^T \\ &= B_{b,n}(P_{m,n} \otimes I_b)(I_m \otimes F_b) \\ &\quad \mathcal{P}_{m,n}^T X \mathcal{P}_{m,n}(I_m \otimes F_b)(P_{m,n}^T \otimes I_b) B_{b,n}^T \end{aligned}$$

The algorithmic variant expressed in this equation first evaluates $\mathcal{H}_{m,n} = \mathcal{P}_{m,n}^T X \mathcal{P}_{m,n}$ and then finds the 2D FFT of the resulting $b \times b$ blocks. The blocks are then permuted in bit-reversed order, before applying the butterfly operations in $B_{b,n}$. The effect has been to split the permutations $\Pi_{b,n}^T X \Pi_{b,n}$ in Eq. 19 into two simpler permutation operations.

Performance Experiments

The run-time performance of the Cholesky factorisation and the recursive matrix multiplication and 2D FFT algorithms for Morton order arrays, described in the preceding sections, has been compared with canonical non-recursive algorithms on two different computing platforms. In these experiments the input arrays were taken to be $n \times n$, where n is an exact power of 2. The block size, b, at which the recursion terminated was varied for each matrix size. b is also the minimum block size used in the Morton ordering of the matrix. Each reported time is the average over 10 separate program executions. The standard deviation for each time was also found and in all cases was less than 2% of the average time. In all cases the input and output matrices are in Morton order, and the timings reported are for the algorithms described in the previous sections.

In the performance experiments presented below there is no direct programmatic control over the content of the cache; this is determined by the caching policies of the different platforms. Being able to explicitly control the movement of data in and out of the cache might be expected to further improve the performance of Morton order algorithms since it would then be possible to ensure that both blocks were in cache before multiplying them.

Platform 1: Intel Core i7

The first computing platform is a MacBook Pro with a 2.5Gz Intel Core i7 processor. This processor has four cores, with a 256 KB L2 cache per core, and a 6 MB L3 cache. The system has a 16GB main memory, and the operating system is OS X 10.10.4. Version 4.8.2 of the gcc compiler was used with the "O3" optimisation flag set.

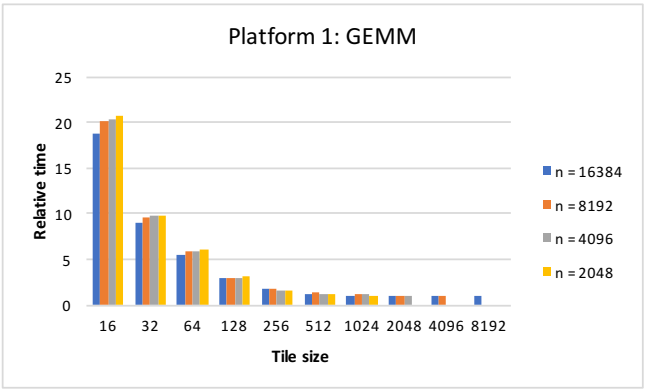
Platform 2: Xeon E5-2620

The second platform, named g00, is a node with two sockets, each containing a 2GHz Intel Xeon E5-2620 processor. This processor has 6 cores with a 256 KB L2 cache per core, and a 15MB L3 cache. The system has a 16 GB main memory and the operating system is Red Hat Enterprise Linux Server release 6.2. Version 4.8.5 of the gcc compiler was used with the "O3" optimisation flag set.

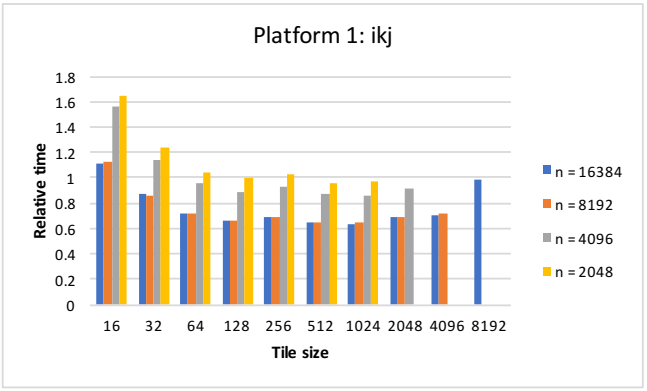
Performance Results for Matrix Multiplication

The recursive matrix multiplication algorithm shown in Alg. 2 multiplies $2^{t-r} \times 2^{t-r}$ matrix tiles at the leaves of

the recursion tree. The algorithm chosen to do this, and its implementation, has a large impact on the performance of Alg. 2. The issues involved in designing algorithms for high performance matrix multiplication have been discussed by Goto and Van de Geijn[7] who point out the importance of a layered approach based on a small number of highly optimized kernels, and the efficient use of L2 cache and the Translation Look-aside Buffer. These kernels may be written in assembly code. The impact of the choice of routine for the matrix multiplication of the tiles is shown in Fig. 3, which presents results for platform1 using (a) the BLAS matrix multiplication routine _GEMM, and (b) a reference implementation making use of an ikj loop ordering that ensures unit stride in accessing the matrices. The matrix elements are stored as 4-byte floating-point values.



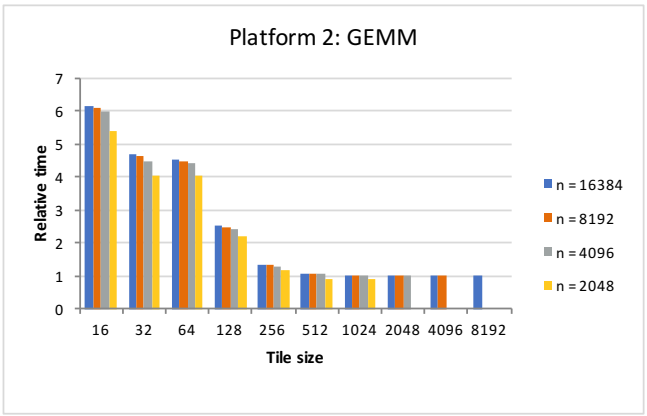
(a)



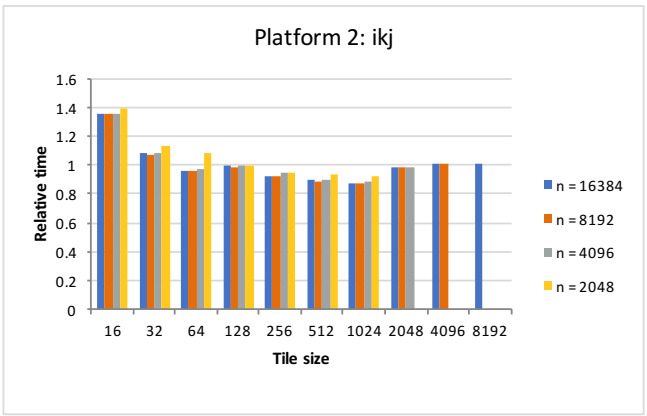
(b)

Figure 3. Matrix multiplication using (a) GEMM and (b) a reference ikj implementation: time for tiled Morton order algorithm relative to untiled row-major order algorithm on Platform 1.

Figure 3(b) shows that Morton ordering reduces the runtime by over 35% in some cases when the reference ikj algorithm is used. For a small block size the Morton ordering case is slower than the untiled RM case, possibly because the loops are shorter, and for larger blocks sizes Morton ordering does not give any significant advantage for matrices smaller than 4096×4096. However, the performance benefits of



(a)



(b)

Figure 4. Matrix multiplication using (a) GEMM and (b) a reference ikj implementation: time for tiled Morton order algorithm relative to untiled row-major order algorithm on Platform 2.

Morton ordering can be seen for larger matrices and block sizes $b \geq 32$, as the improved cache performance outweighs the effect of the shorter loops. Once the blocks no longer fit into L3 cache, then the performance of the Morton order case worsens. Figure 3(a) shows that for small blocks the tiled Morton ordering case reduces the performance by up to a factor of about 20. For block sizes greater than 1024 the performance of the tiled and untiled algorithms is comparable, with the tiled algorithm being 5% faster for $n = 16384$ and $b = 4096$.

The timings results for platform 2, shown in Fig. 4, exhibit similar behaviour to those in Fig. 3. However, the performance improvement of the tiled Morton ordered algorithm is less marked than for platform 1. For matrices larger than 2048×2048 and block sizes b between 256 and 1024 the tiled Morton order algorithm reduces the execution time by up to 13%. There is no performance advantage for block sizes of $b = 2048$ or larger, and again this may be attributed to the fact that for such block sizes the input matrix blocks do not fit in the L3 cache.

Performance Results for Cholesky Factorisation

The relative timings for Cholesky factorisation for platform 1 are shown in Fig. 5. For each size of matrix times are shown relative to the time to perform the factorisation using the LAPACK routine `DPOTRF`, and it should be noted that this routine uses a blocked algorithm with the block size automatically chosen according to the matrix size. Figure 5(a) shows results for a matrix in row-major order using a blocked algorithm that calls `DPOTRF`, `DTRSM`, and `DSYRK` to perform the main steps of Alg. 3. Figure 5(b) also shows results for a row-major matrix, but for a tiled algorithm constructed using LAPACK and BLAS routines. Finally, Fig. 5(c) shows results for a tiled algorithm, but for a Morton order matrix.

Comparison of Figs. 5(a) and 5(b) shows that for a row-major matrix the blocked algorithm is faster than the tiled algorithm with the same block size. However, Figs. 5(b) and 5(c) show that Morton ordering gives some improvement over the row-major case for all matrix sizes. Similar behaviour is seen in the timing results for platform 2, shown in Fig. 6. Figures 5 and 6 show that for a given block size, the time relative to `DPOTRF` is smaller for smaller matrices. This is because `DPOTRF` runs more efficiently for larger matrices, as may be seen in Fig. 8, which shows the time per floating-point operation as a function of matrix size, n, assuming the number of floating-point operations for Cholesky factorisation is $n^3/3$.

Figure 7 shows the relative times for the binary recursive Cholesky factorisation algorithm, given in Alg. 4, for Platform 1. Figure 7(a) shows that the binary recursive algorithm is faster than `DPOTRF` for all matrix and tile sizes presented. It can also be seen that Morton order gives some performance advantage over RM order for sufficiently small tile sizes, but there is not much difference at larger tile sizes. Similar results were found for Platform 2, and so are not shown, although the relative performance of `DPOTRF` was better in this case.

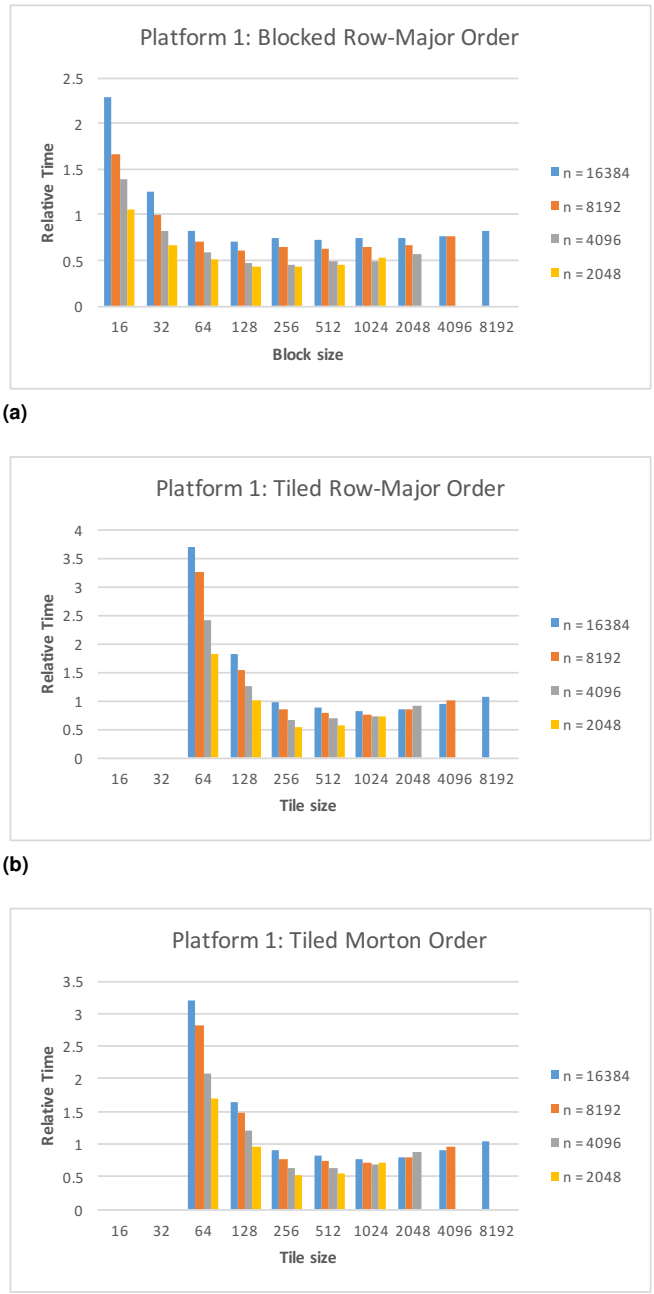


Figure 5. Cholesky factorisation using (a) a blocked algorithm and RM order, (b) a tiled algorithm and RM order, and (c) a tiled algorithm and Morton order on Platform 1. All times are relative to the time taken by the LAPACK routine DPOTRF.

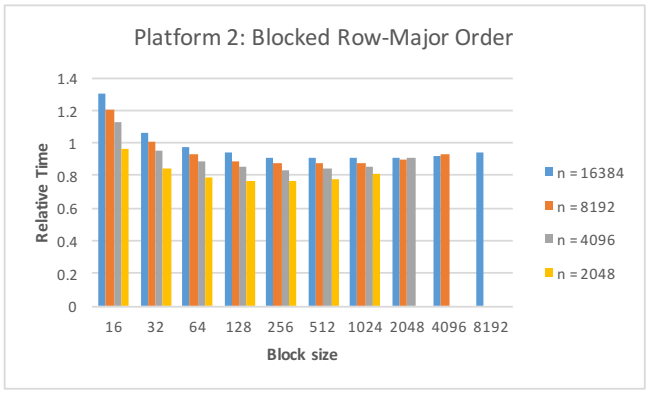
Performance Results of FFT

Two main options were considered for performing the FFTs in the leaves of the recursion tree, i.e., the routine `fft2D` in the Alg. 11:

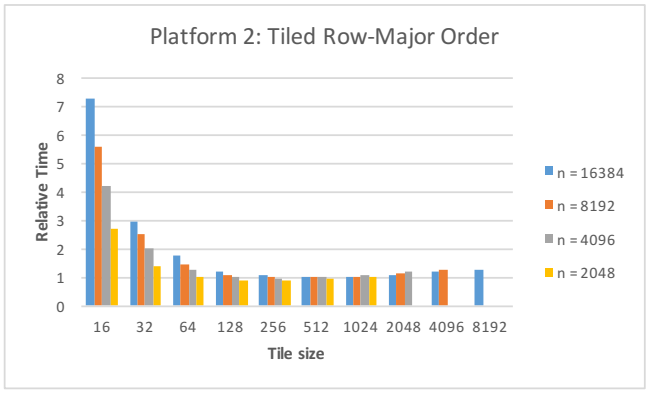
1. The transpose based Alg. 5 that maintains unit stride and make use of multiple one-dimensional FFTs.
2. The two-dimensional vector-radix routine, Alg. 6.

In addition, different ways of evaluating the dilated integers needed to index Morton ordered arrays were considered. Timing results are shown for platforms 1 and 2 in Figs. 9 and 10, respectively. In all cases, the `fft2D`, `butterflyPre`, and `butterflyPost` routines in Algs. 9 and 11 were implemented in the C language.

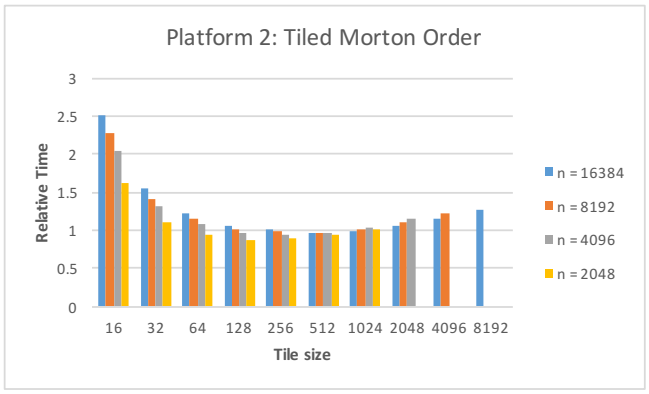
Figure 9(a) shows that for a given array size the relative time for the vector radix case tends to be larger for smaller tiles, decreases as the tile size increases, and then begins to rise again as the tile size increases further. A similar trend can be seen in Fig. 9(b), except that for large arrays no increase in relative time is seen for the larger tile sizes. The vector-radix algorithm involves $O(n^2 \log n)$ large stride



(a)

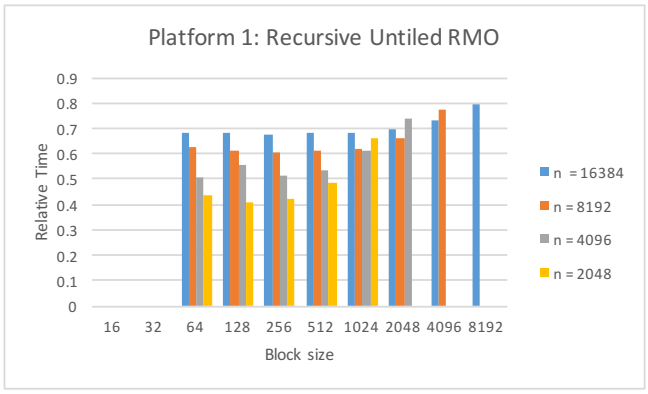


(b)

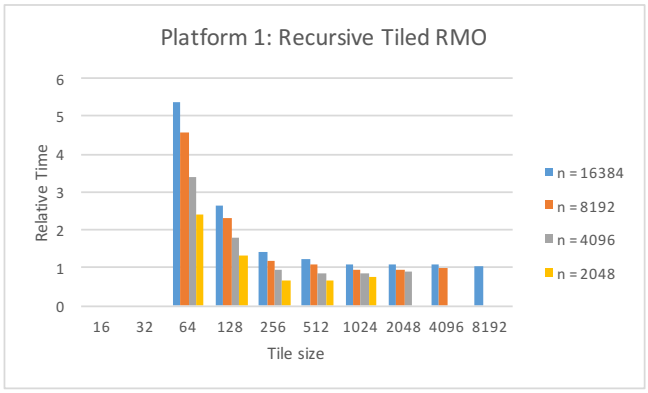


(c)

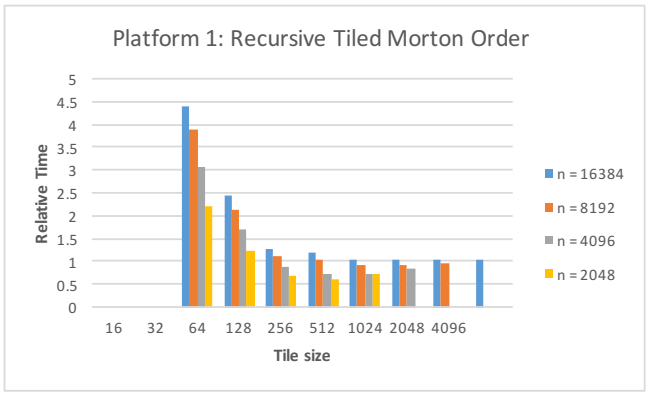
Figure 6. Cholesky factorisation using (a) a blocked algorithm and RM order, (b) a tiled algorithm and RM order, and (c) a tiled algorithm and Morton order on Platform 2. All times are relative to the time taken by the LAPACK routine DPOTRF.



(a)



(b)



(c)

Figure 7. Cholesky factorisation using the binary recursive algorithm: (a) a blocked algorithm and RM order, (b) a tiled algorithm and RM order, and (c) a tiled algorithm and Morton order on Platform 2. All times are relative to the time taken by the LAPACK routine DPOTRF.

array accesses, which impacts performance once a tile does not fit into cache. The transpose-based algorithm maintains unit stride access in the FFT computation at the added expense of having to perform two array transposes, which requires $O(n^2)$ data movements that in general involve non-unit stride accesses. Thus, for large tiles the performance of the vector-radix algorithm is degraded more by cache misses than the transpose-based algorithm. This is evident from Fig. 11, which shows the times per flop for the vector-radix and transpose-based 2D FFT algorithms on platforms 1 and 2, assuming that the number of floating-point operations to do a complex $n \times n$ FFT is $10n^2 \log_2 n$. The time per flop increases with array size, n, for the vector-radix algorithm, whereas it is almost constant for the transpose-based algorithm.

Discussion

Morton ordering is expected to be most effective if most of the floating-point operations in an algorithm are performed in the leaves of the recursion tree because this maximises the ratio of computation to data movement (assuming the

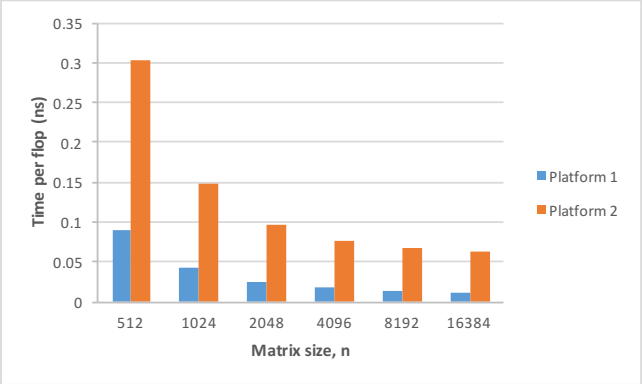
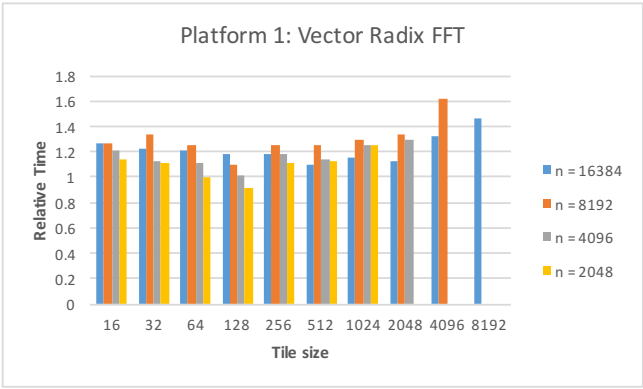
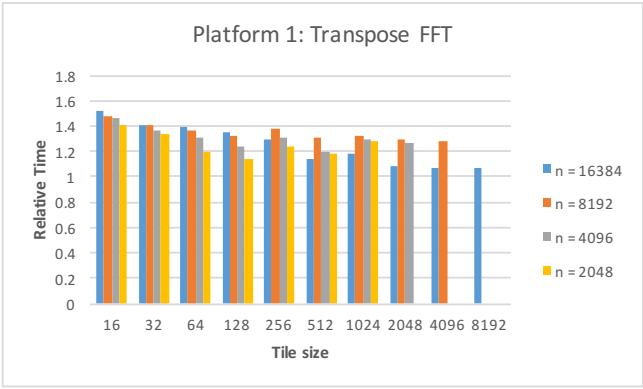


Figure 8. Cholesky factorisation: dependence of time per floating-point operation on matrix size, n.

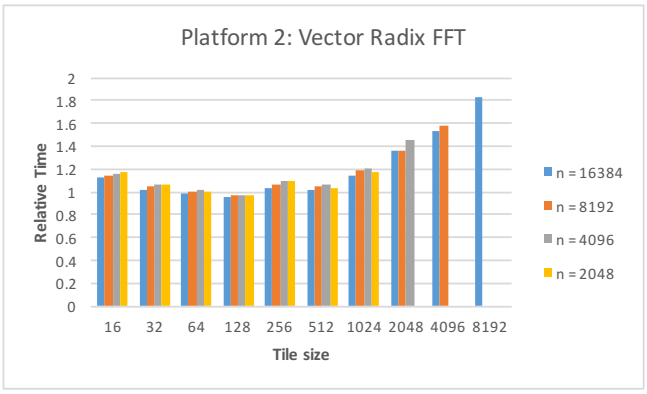


(a)

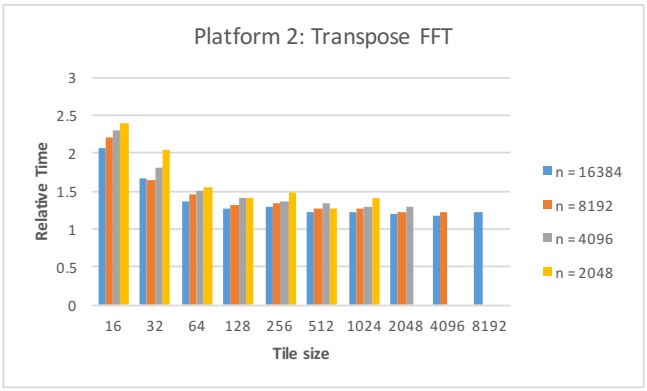


(b)

Figure 9. Platform 1: FFT for Morton order arrays using (a) the vector radix algorithm, and (b) the standard transpose-based algorithm, for performing the FFTs in the leaves of the recursion tree. In both cases times are relative to the time for a standard transpose-based algorithm on a RM array of the same size.



(a)



(b)

Figure 10. Platform 2: FFT for Morton order arrays using (a) the vector radix algorithm, and (b) the standard transpose-based algorithm, for performing the FFTs in the leaves of the recursion tree. In both cases times are relative to the time for a standard transpose-based algorithm on a RM array of the same size.

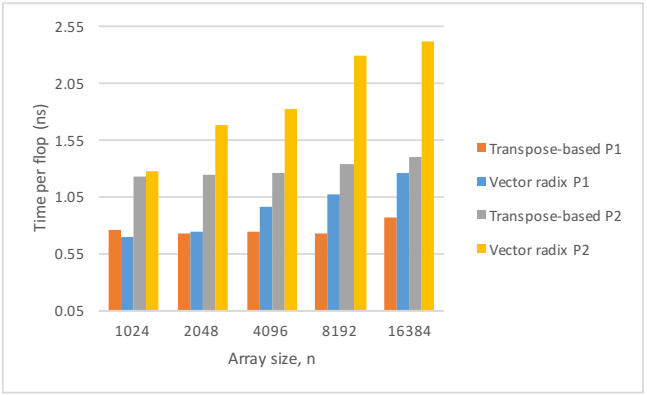


Figure 11. Dependence of time per floating-point operation on array size, n, for the vector-radix and transpose-based FFT algorithms.

tiles involved in the computation fit into higher-level memory such as the L3 cache). In the tiled algorithm for matrix multiplication, shown in Alg. 2, all the computation is done in the leaves of the recursion tree, and the total number of floating-point operations performed is $2n^3$.

In contrast, the algorithm for block Cholesky factorisation in Alg. 3 exhibits tail recursion, which most modern compilers will convert to an iterative algorithm to avoid the overhead of allocating a new stack frame on each recursive call. Thus, whereas for matrix multiplication recursion provided a natural and simple way of expressing the Morton order algorithm, there is no such advantage for Cholesky factorisation. The number of floating-point operations involved in each phase of Alg. 3 is as follows:

1. Cholesky factorisation of A_{00}: $b^3/3$ flops.
2. Triangular solve of $L_{10}L_{00}^T = A_{10}$: mb^2 flops, where m is the number of rows of L_{10} and A_{10}.

3. Symmetric rank-b update $L_{11}L_{11}^T = A_{11} - L_{10}L_{10}^T$: $mb(m+b)/2$ flops.

Summing these expressions over the n/b stages of the algorithm gives a total flop count of:

$$T_{CF}(n,b) = \frac{n^3}{3} + \frac{bn(n-b)}{2} \tag{29}$$

In the tiled implementation used in this work, the routine `DGEMM` is used to multiply the tiles when doing the symmetric rank-b update. This means that when updating the diagonal tiles extra work is done to update the elements above the diagonal, which accounts for the second term in Eq. 29. This extra work could be avoided by using `DSYRK` to update the diagonal tiles; however, it was found that although this improves the performance for larger tile sizes, it results in a small reduction in performance for smaller tile sizes.

In the tiled Algs. 9 and 11 for performing a $2^t \times 2^t$ FFT on a 2D Morton ordered array, there are 4^k nodes at level k of the recursion tree. Each leaf node computes a $b \times b$ FFT, where $b = 2^{t-r}$. This involves $10b^2 \log_2 b$ floating-point operations. Thus, the number of floating-point operations associated with the leaf nodes is:

$$10(2^t)^2(t-r) \tag{30}$$

From Eqns. 15 and 17, non-leaf node multiplies four $2^{t-k-1} \times 2^{t-k-1}$ matrices by a diagonal matrix, and does eight matrix additions. Since, in general, the matrices are complex, the number of floating-point operations associated with a non-leaf node is $40 \times (2^{t-k-1})^2$, and the total number for all the non-leaf nodes in the recursion tree is:

$$40 \sum_{k=0}^{r-1} 4^k \times 2^{2t-2k-2} = 10 \times (2^t)^2 r \tag{31}$$

It can be seen that the total number floating-point operations is, as expected, $10 \times n^2 \log_2 n$, where $n = 2^t$. However, the ratio of non-leaf to leaf flops is $r/(t-r)$. For the matrix multiplication algorithm the corresponding ratio is zero. Thus, in the FFT case relatively more computational work is done in the higher levels of the recursion tree, which explains why Morton ordering is less effective in improving the performance of the FFT algorithm when compared with matrix multiplication.

It might be expected that the recursive algorithms presented here would incur overheads that are not applicable in loop-based algorithms. The number of recursive calls in the Cholesky factorisation algorithm is $O(n/b)$. For the recursive 2D FTT and matrix multiplication algorithms the number of recursive calls is $O(n^2/b^2)$ and $O(n^3/b^3)$, respectively. Thus, if recursive overhead has a large impact on performance this should be more apparent for large values of n/b in the matrix multiplication timings.

Summary and Conclusions

The timing results presented here show that, for the three algorithms considered, Morton ordering of arrays results in optimal performance for intermediate tile sizes of about 256×256. For smaller tile sizes performance decreases due to the overhead associated with managing recursion, such as stack frame allocation. For larger tile sizes the tiles no longer fit into cache, which again degrades performance. In some cases it was found that a tiled algorithm based on Morton ordering has higher performance than the corresponding canonical implementation, although this is dependent on the algorithm and hardware. The efficient use of the Translation Look-Aside Buffer (TLB) can also have a significant effect on performance, as demonstrated by Park et al.[13].

In the timing experiments presented here no attempt has been made to explicitly control the transfer of data between different levels in the memory hierarchy – this is under the control of the run time system and possibly the compiler. It could be argued that tiled algorithms using Morton ordering would have even better performance if data movement were controlled more at the program level. This idea of "programming the memory hierarchy" underlies the Sequoia programming language developed at Stanford University[2;6]. Sequoia represents the memory hierarchy directly in the programming model and provides abstractions that separate the expression of algorithms from machine-dependent optimisation. The lessons learned from Sequoia have now been carried forward into the Legion programming model and runtime system[3], which allows the programmer to achieve good performance through reasoning about data locality and task independence. Similar ideas have been put forward by Schneider et al.[14], who compare the use and performance of the Cellgen and Sequoia programming models with the Cell SDK for two applications running on the Cell Broadband Engine processor.

Support for tiled algorithms has been investigated by Bikshandi et al.[4], based on their Hierarchically Tiled Array (HTA) datatype, of which Morton ordering is a special case. Bikshandi et al. point out that in many cases the HTA approach facilitates the expression of parallelism, and demonstrate efficient parallel HTA implementations of several algorithms. Similarly, the recursive matrix multiplication and FFT algorithms for Morton order arrays presented in Algs. 2 and 11 are readily parallelisable as

each recursive call can be made independently. However, in the recursive Cholesky factorisation algorithm in Alg. 4 the triangular solve and symmetric update tasks must be executed in order between the recursive calls to `choleskyBinaryRecursive`, so the latter cannot be executed in parallel. The parallelisation of recursive algorithms applied to Morton order arrays on modern multicore and manycore processors will be presented in a subsequent paper.

Acknowledgements

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

1. Anderson E, Bai Z, Bischof C, Blackford L, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A and Sorensen D (1999) *LAPACK Users' Guide*. Third edition. Society for Industrial and Applied Mathematics. DOI:10.1137/1.9780898719604. URL http://epubs.siam.org/doi/abs/10.1137/1.9780898719604.
2. Bauer M, Clark J, Schkufza E and Aiken A (2011) Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia. In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11. New York, NY, USA: ACM. ISBN 978-1-4503-0119-0, pp. 13–24. DOI:10.1145/1941553.1941558. URL http://doi.acm.org/10.1145/1941553.1941558.
3. Bauer M, Treichler S, Slaughter E and Aiken A (2012) Legion: Expressing locality and independence with logical regions. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press. ISBN 978-1-4673-0804-5, pp. 66:1–66:11. URL http://dl.acm.org/citation.cfm?id=2388996.2389086.
4. Bikshandi G, Guo J, Hoeflinger D, Almasi G, Fraguela BB, Garzarán MJ, Padua D and von Praun C (2006) Programming for parallelism and locality with hierarchically tiled arrays. In: *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06. New York, NY, USA: ACM. ISBN 1-59593-189-9, pp. 48–57. DOI:10.1145/1122971.1122981. URL http://doi.acm.org/10.1145/1122971.1122981.
5. Dongarra JJ, Du Croz J, Hammarling S and Duff IS (1990) A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16(1): 1–17. DOI:10.1145/77626.79170. URL http://doi.acm.org/10.1145/77626.79170.
6. Fatahalian K, Horn DR, Knight TJ, Leem L, Houston M, Park JY, Erez M, Ren M, Aiken A, Dally WJ and Hanrahan P (2006) Sequoia: Programming the memory hierarchy. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06. New York, NY, USA: ACM. ISBN 0-7695-2700-0. DOI:10.1145/1188455.1188543. URL http://doi.acm.org/10.1145/1188455.1188543.
7. Goto K and van de Geijn RA (2008) Anatomy of high performance matrix multiplication. *ACM Transactions on Mathematical Software* 34(3): 12:1–12:25. DOI:10.1145/1356052.1356053. URL http://doi.acm.org/10.1145/1356052.1356053.
8. Gustavson FG (1997) Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development* 41(6): 737–755. DOI:10.1147/rd.416.0737.
9. Gustavson FG and Walker DW (2014) Algorithms for in-place matrix transposition. In: Wyrzykowski R, Dongarra J, Karczewski K and Waśniewski J (eds.) *Parallel Processing and Applied Mathematics: 10th International Conference, PPAM 2013, Warsaw, Poland, September 8-11, 2013, Revised Selected Papers, Part II*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-55195-6, pp. 105–117. DOI:10.1007/978-3-642-55195-6_10. URL http://dx.doi.org/10.1007/978-3-642-55195-6_10.
10. Loan CV (1992) *Computational Frameworks for the Fast Fourier Transform*. SIAM Press.
11. Mellor-Crummey J, Whalley D and Kennedy K (2001) Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming* 29(3): 217–247. DOI:10.1023/A:1011119519789. URL http://dx.doi.org/10.1023/A:1011119519789.
12. Morton GM (1966) A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd.
13. Park N, Hong B and Prasanna VK (2003) Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems* 14(7): 640–654. DOI:10.1109/TPDS.2003.1214317.
14. Schneider S, Yeom JS and Nikolopoulos DS (2009) Programming multiprocessors with explicitly managed memory hierarchies. *Computer* 42(12): 28–34. DOI:10.1109/MC.2009.407.
15. Thiyagalingam J, Beckmann O and Kelly PHJ (2006) Is Morton layout competitive for large two-dimensional arrays yet? *Concurrency and Computation: Practice and Experience* 18(11): 1509–1539. DOI:10.1002/cpe.v18:11. URL http:

//dx.doi.org/10.1002/cpe.v18:11.

16. Valsalam V and Skjellum A (2002) A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience* 14(10): 805–839. DOI:10.1002/cpe.630. URL http://dx.doi.org/10.1002/cpe.630.
17. Wise DS and Raman R (2008) Converting to and from dilated integers. *IEEE Transactions on Computers* 57(4): 567–573. DOI:10.1109/TC.2007.70814.